VxBPEL

Koning, Michiel; Sun, Chang-ai; Sinnema, Marco; Avgeriou, Paris

Published in:
Information and Software Technology

DOI:
10.1016/j.infsof.2007.12.002

Publication date:
2009

Available online at www.sciencedirect.com
ScienceDirect
Information and Software Technology 51 (2009) 258–269
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

VxBPEL: Supporting variability for Web services in BPEL ☆

Michiel Koning [a], Chang-ai Sun [b,*], Marco Sinnema [a], Paris Avgeriou [a]

[a] Department of Computer Science, University of Groningen, P.O. Box 800, 9700 AV Groningen, The Netherlands
[b] School of Computer and Information Technology, Beijing Jiaotong University, 100044 Beijing, PR China

Received 15 January 2007; received in revised form 5 November 2007; accepted 27 December 2007
Available online 26 January 2008

Abstract

Web services provide a way to facilitate the business integration over the Internet. Flexibility is an important and desirable property of Web service-based systems due to dynamic business environments. The flexibility can be provided or addressed by incorporating variability into a system. In this study, we investigate how variability can be incorporated into service-based systems. We propose a language, VxBPEL, which is an adaptation of an existing language, BPEL, and able to capture variability in these systems. We develop a prototype to interpret this language. Finally, we illustrate our method by using it to handle variability of an example.

Keywords: Variability; Web service; Service-based system; Business Process Execution Language

1. Introduction

Web services have evolved as a means to integrate processes and applications at an inter-enterprise level [19]. Consider the travel agency where one would like to book a flight, hotel and car rental at the same time; the online store where one can see the current stock for the item one wants to buy; the supermarket that automatically places an order at the distributor when stocks run low. Web services are special software components that are located, bound and executed at run-time and can provide a solution to allow such systems to interact in a seamless way using standard internet protocols, such as UDDI, WSDL and SOAP [11].

A Web service consists of two parts. One is the software which implements the actual functionality. The other part is an interface specifying this functionality, defined by WSDL. Several Web services can be combined to form a new system. Such a system can be seen as a composite Web service which usually implements a business process.

A system built on services is called a service-centric system (SC system). Orchestration [22] is widely used to describe executable business processes with interactions with possibly both internal and external Web services. Services in an SC system are highly reusable and allow fast adaptation to changing requirements. At run-time, Web services can be bound to different concrete service implementations. This means that SC systems offer extreme flexibility.

Web services exist in a dynamic environment. It is possible that services become less available or unavailable due to fluctuations in available bandwidth and throughput rates. Because Web service providers are usually bound by a contract called an SLA (Service Level Agreement) to provide a certain level of Quality of Service (QoS), with penalties for deviating from the agreed upon QoS, such irregularities can have negative consequences. The ability of defining variability in an SC system presents the following advantages.

– It can enhance system availability, by replacing an unavailable service by another.

☆ The work was done when the second author worked as a postdoctoral fellow at the University of Groningen.

* Corresponding author.
E-mail addresses: casun@bjtu.edu.cn (C.-a. Sun), p.avgeriou@cs.rug.nl (P. Avgeriou).

0950-5849/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2007.12.002

– It supports run-time reconfiguration. Rebinding of services can be done at run-time.
– It helps meet the agreed upon QoS, by optimising performance through service replacement if necessary.
– It can be used to optimise quality attributes by changing the configuration of the system.

However, how to model variation in SC systems is omitted in existing service composition approaches. These approaches [5,10,23,27,28] support self-reconfiguration and automated composition in SC systems and allow (automated) rediscovery and rebinding of Web services, but do not allow arbitrary parts of their systems to be variable and do not allow defining several configurations between which can be switched arbitrarily.

Recently, several attempts on extension to BPEL are reported to improve its modularity or support adaptation of business processes defined by BPEL [7–9,12–16]. The approaches presented in [7–9,12] employ aspect-oriented programming technique to address additional concerns in business processes, which effectively solve the scatter problem and the tangling problem with extension of BPEL. The approaches presented in [13–16] employ the proxy or bus mechanism to explicitly implement the adaptation of business process at run-time at the messaging layer. They do not treat changes as first class entities in the Web service compositions, and thus focus on run-time adaptation in terms of process instances.

In this study, we propose an approach, VxBPEL, to deal with variability in SC systems. It allows one to define variation points, variants and configurations for a process in an SC system. VxBPEL addresses the adaptive composition of Web services by providing the variability constructs in the language level, and treats the changes as first-class entities which are omitted in most current work, particularly those focusing on process instances in the implementation level. The specifications of adaptive Web services composition in VxBPEL clearly integrate main business logic and adaptation of process elements. The adaptation is optional naturally since the designers are free to use variability constructs. The execution of adaptation is supported at compile time and at runtime since the extensions to the BPEL engine are used to interpret the variability constructs.

The paper is organized as follows. Section 2 introduces some underlying concepts and technologies. Section 3 presents VxBPEL, an extension to the BPEL language and a prototype implementation to interpret VxBPEL. Section 4 discusses a case study for experiments. Section 5 compares our method with related work. Section 6 concludes the paper.

2. Background

In this section, we introduce variability-related concepts, BPEL, which is used to compose an SC system from Web services, and variation modeling of Web services.

2.1. Variability

Increasingly systems have been built out of components [3,18]. Such systems may likely need to be adapted in their life cycle. This can be due to customers' new wishes for the system (i.e., changing requirements), because of compatibility, new developments, etc. In short, it is desirable that changes can be made to the system. It is possible to freely change the internals of such a component, altering its behaviour, as long as the functionality it provides conforms to the interface specification, because its functionality is defined by its interface. It is also possible to change the outward behaviour, by changing the interface (and thus the functionality provided). When the information about the ability to change a system is explicit, it is called "variability"[2].

2.1.1. Variation points and variants

A part of a system that can vary is called a "variation point". Usually, for such a variation several options are defined between which can be chosen, which are called "variants" (or "alternatives"). When such a variant is chosen for each variation point, the collection of these choices is referred to as a "configuration" [6].

2.1.2. Realization relations

Systems can contain variation points at several levels of abstraction. An example is a reservation system. In this system a service is invoked to make a reservation. It can be specified that this service is a variation point and has two variants. This is a high-level view. However, when one looks at the actual implementation of the system, which is a lower-level view, it could be that several variation points are introduced which all have several variants to allow for the option at the higher-level. It could be that the two variants are incompatible with each other, because the two services require different messages, and therefore require extensive changes in the implementation to switch from one variant to another.

Fig. 1 depicts this example. It uses COVAMOF's [24] notation for variation points and realizations: a variation point is denoted by a circle, its associated variants as triangles attached to this circle, and the dashed line separates the VPs on the lower-level from the VP on the higher-level.

In choosing the variant for the reservation service on a higher-level, the variation points on a lower-level (i.e., all the parts that interact with the message that is sent to the service) need to have a certain configuration to realize the higher-level variation point. Such a relation between variation points is called a "realization relation" [24]. A realization relation, when formalized, can however improve the manageability of the variability a system has, because the exact details of which variation points allow which other variation points to exist can then be determined automatically. In other words, one need only be concerned with the variation points at the highest level of abstraction when realization relations are known and formalized.

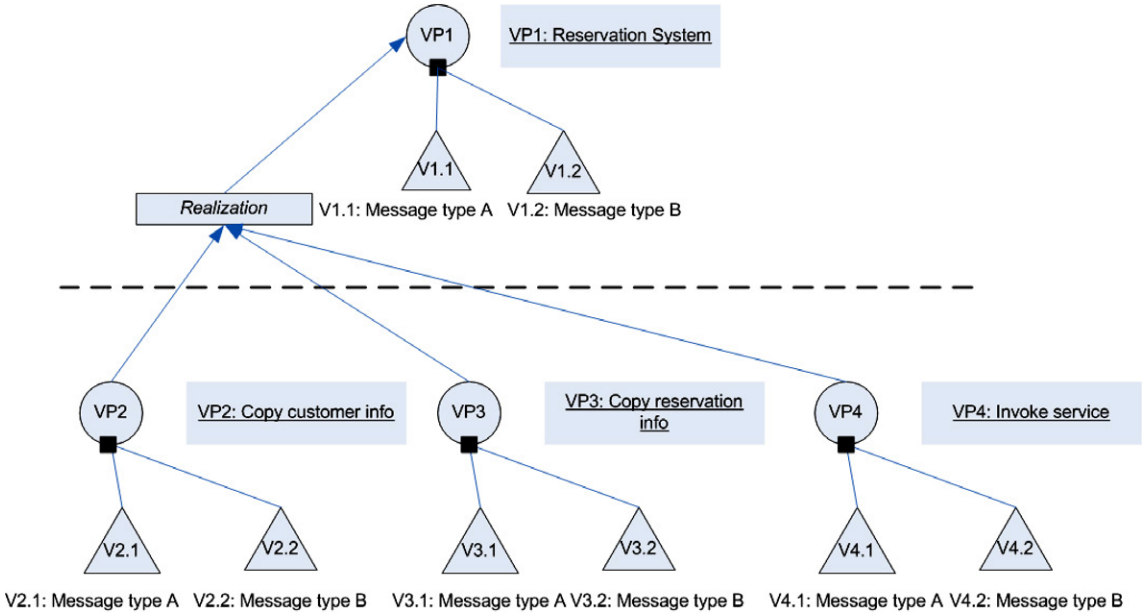

Fig. 1. The choice at the architecture level prescribes choices on the implementation level.

2.1.3. Binding time

There are several stages in the life cycle of a software system where a configuration may be selected, after which it is no longer possible to change the configuration selection. This selection is called "binding", and the stage in the life cycle at which binding occurs is called "binding time". Binding can occur at several stages, e.g. the compilation stage, the installation/deployment stage or at run-time[12,17]. When binding occurs in a certain stage of the life cycle, the system configuration is fixed as of that stage. While run-time binding is easier to implement in the context of Web services due to the separation of their specifications and implementations. When a choice is made at run-time, and binding occurs, it is possible to allow this binding to be redone; in other words, it allows rebinding at run-time. Being able to rebind variation points at run-time means that a system can reconfigure at run-time without shutting down. This is especially interesting for systems in a dynamic environment since this means they can quickly adapt to or respond to changes.

2.2. BPEL

The BPEL language [4,21] provides a notation and semantics for specifying business process behaviour based on Web services. A process is defined in terms of its interactions with partners. A partner may provide services to a process, require services from a process, or interact two-way with a process. BPEL orchestrates Web services by specifying the order in which it is meaningful to call a collection of services, and assigns responsibilities for each of the services to partners. It can be used to specify both the public interfaces for the partners and the description of the executable process. It is an XML-based flow language and it supports structured programming constructs such as if-statements, while-statements, sequence-statements (to execute statements in sequence) and flow-statements (to execute statements in parallel). Since it is focused on service-based business processes, it has native support for the messaging paradigm. Messages may be a type of WSDL or variable types from other namespaces.

Fig. 2 illustrates a simplified example of a BPEL process. A BPEL process consists of activities (such as

```
<process name="loanApprovalProcess">
 <sequence>
  <receive partnerLink="customer"
        operation="request" .../>

  <invoke partnerLink="assessor"
   operation="check"
   inputVariable="request"
   outputVariable="risk"/>

  <assign>
   <copy>
    <from>
     <expression>'yes'</expression>
    </from>
    <to variable="approval"
     part="accept"/>
   </copy>
  </assign>

  <reply partnerLink="customer"
       operation="request"
    variable="approval"/>
 </sequence>
</process>
```

Fig. 2. A simplified example of a BPEL process definition.

⟨invoke⟩ and ⟨assign⟩), which are basically execution steps, and activity containers (such as ⟨sequence⟩), which provide information about the execution of the contained activities. Fig. 2 also shows how BPEL supports message exchanges between activities. Messages are basically treated as complex variables. The ⟨receive⟩, ⟨invoke⟩ and ⟨reply⟩ activities are all message-related statements. There are also control-flow activities such as if-statements and while-statements and variable-related activities such as the ⟨assign⟩ activity in the example.

2.3. Variability in SC Systems

Several types of variability need to be captured in variation points of a service composition, in order to model variability in Web services. These types are [26]:

- Replacing a service by a different one with the same interface.
- Replacing a service by one with a different interface.
- Changing the parameters with which a service is invoked.
- Changing the composition of the system.

When we model variability of Web services in service compositions, it is necessary to capture the above types of variability and at the same time realization relations between variation points. In this study, we assume that BPEL is used to define a service composition and we investigate how to extend this language to support modeling of variability.

3. VxBPEL: enabling variability modeling in BPEL

SC systems are usually process-driven, and these processes are defined by a process description language. In order to capture variation point and dependency information in an SC system, the process description language used for such a system must be extended to allow the definition of variability information.

```
<vxbpel:VariationPoint name="VP1">
  <vxbpel:Variants>
    <vxbpel:Variant name="default">
      <vxbpel:VPBpelCode>
        <invoke .../>
      </vxbpel:VPBpelCode>
    </vxbpel:Variant>
    <vxbpel:Variant name="alternative1">
      <vxbpel:VPBpelCode>
        <sequence ...>
          ...
        </sequence>
      </vxbpel:VPBpelCode>
    </vxbpel:Variant>
  </vxbpel:Variants>
</vxbpel:VariationPoint>
```

Fig. 3. Definition of a variation point in VxBPEL.

3.1. The extension: VxBPEL

BPEL is a widely accepted language for this purpose. VxBPEL is an extension to the standard BPEL language, which introduces new activities (keywords) that allow variability information, specifically variation points, variants and realization relations, to be modelled. To include variability information in the BPEL process, BPEL elements are enclosed by new VxBPEL elements (recognizable in the examples by the XML namespace prefix vxbpel). By defining it in this way, it is possible to see which parts of the process are variable merely by looking at the process definition.

To indicate that a part of a BPEL process is a variation point, it is enclosed by a ⟨VariationPoint⟩ element. Variants defined for this variation point are listed within such an element by several ⟨Variant⟩ elements, enclosed

```
<process>
  <flow>
    <!-- process contents -->
  </flow>

  <vxbpel:ConfigurableVariationPoints>
    <vxbpel:ConfigurableVariationPoint
       id="CVP1" defaultVariant="default">
      <vxbpel:Name>The name goes here.
      </vxbpel:Name>
      <vxbpel:Rationale>
        ...
      </vxbpel:Rationale>
      <vxbpel:Variants>
        <vxbpel:Variant name="default">
          <vxbpel:VariantInfo>
            <!-- Information that pertains
                 only to this variant. -->
          </vxbpel:VariantInfo>
          <vxbpel:RequiredConfiguration>
            <vxbpel:VPChoices>
              <vxbpel:VPChoice
                vpname="VP1"
                variant="default"/>
            </vxbpel:VPChoices>
          </vxbpel:RequiredConfiguration>
        </vxbpel:Variant>
        <vxbpel:Variant
             name="alternative1">
          <vxbpel:VariantInfo>
            ...
          </vxbpel:VariantInfo>
          <vxbpel:RequiredConfiguration>
            <vxbpel:VPChoices>
              <vxbpel:VPChoice
                vpname="VP1"
                variant="alternative1"/>
            </vxbpel:VPChoices>
          </vxbpel:RequiredConfiguration>
        </vxbpel:Variant>
      </vxbpel:Variants>
    </vxbpel:ConfigurableVariationPoint>
  </vxbpel:ConfigurableVariationPoints>

</process>
```

Fig. 4. Definition of a configurable (high-level) variation point, including the realization relations.

by a ⟨Variants⟩ container element. Each of these variants has a name as indicated by the name attribute, and associated BPEL code to be placed in the process definition, defined by the ⟨BpelCode⟩ element. These variation points can be placed inside a BPEL process in any place where a single activity (such as ⟨invoke⟩) or activity container (such as ⟨sequence⟩) can be placed. An example can be seen in Fig. 3.

Because there may be many of these variation points throughout a BPEL process and they will often not be isolated from each other, it is also possible to capture higher-level variation points which describe the relations between the variation points inside the process. In VxBPEL, these are called "configurable variation points" and are contained in ⟨ConfigurableVariationPoint⟩ elements. Each of these configurable variation points also has variants, ⟨Variant⟩, enclosed in the ⟨Variants⟩ element, and for each of these variants an element ⟨RequiredConfiguration⟩ exists, which indicate for each high level variant what lower-level variants need to be selected through a number of ⟨VPChoice⟩ elements. In other words, these high-level variation points cover realization relations. The only variation points that should be actively selected are these, as then the lower-level variation points will automatically be set accordingly. To help the user (or a process that automates process reconfiguration) select the correct variant, information is added about the variation points and the variants in the ⟨Rationale⟩ and ⟨VariantInfo⟩ elements. If this information is formalized, automatic configuration is possible. The initial configuration of each configurable variation point must be defined through the defaultVariant attribute.

The configurable variation points are defined inside a process definition. Fig. 4 shows that these configurable variation points are defined in a container just before the end tag of the process (⟨/process⟩), namely ⟨ConfigurableVariationPoints⟩.

3.2. Supporting various types of variability modeling

The idea behind the VxBPEL extensions was to model variability generically. That is, VxBPEL was designed to be able to model all these types in the same way and thus have more flexibility. We briefly discuss how to model each type of variability with VxBPEL.

3.2.1. Service replacement

This actually covers both the first (replacing a service by one with the same interface) and second (replacing a service by one with a different interface) type. Although BPEL itself allows services with identical interfaces to be bound at run-time, it is conceivable one wants to define explicitly which service is to be used for which configuration of the system. In that case, an extra partner link could be added for each variant, and each of these variants would call a different service. In VxBPEL:

```
<vxbpel:VariationPoint name="VPService1">
  <vxbpel:Variants>
    <vxbpel:Variant name="Service1A">
      <vxbpel:VPBpelCode>
        <invoke ... partnerLink="service1a"/>
      </vxbpel:VPBpelCode>
    </vxbpel:Variant>
    <vxbpel:Variant name="Service1B">
      <vxbpel:VPBpelCode>
        <invoke ... partnerLink="service1b"/>
      </vxbpel:VPBpelCode>
    </vxbpel:Variant>
  </vxbpel:Variants>
</vxbpel:VariationPoint>
```

As the actual interface for a service is captured in the definition of the partnerLink, one can see that modeling a variation point as such means that it is possible to define both invoke statements with different values for the partnerLink parameters, defined elsewhere, and thus allowing both types of variability to be captured. Note that it is possible for both services to have different input and output variables, in which case the surrounding statements which prepare a message for sending will also need to be adapted.

3.2.2. Service parameters

This type of variability is modelled similarly to the previous type. However, it is dependent on how the parameters for this service need to be set: either by altering the message sent to this service, or by first invoking a different operation of a service in order to set parameters for a next request. Surrounding statements will need to be adapted, in the first case by an invoke statement to call a different operation, or in the second case by an assign statement to change the message contents. Suppose a service is normally called without setting parameters beforehand (i.e., using the default settings):

```
<invoke inputVariable="input"
      name="invService"
      operation="op" outputVariable="output"
      partnerLink="serviceName"
      portType="..."/>
```

and one wants to be able to set parameters for a service first, by invoking an operation that sets the service parameters:

```
<sequence name="setserviceparams">
 <invoke inputVariable="par"
       name="invParams"
       operation="setParams"
       outputVariable="setParamsOut"
       partnerLink="serviceName"
       portType="..."/>
 <invoke inputVariable="input"
       name="invService"
       operation="op" outputVariable="output"
       partnerLink="serviceName"
       portType="..."/>
</sequence>
```

one would then model a variation point thus:

```
<vxbpel:VariationPoint name="confService">
  <vxbpel:Variants>
    <vxbpel:Variant name="noParams">
      <vxbpel:VPBpelCode>
        <invoke inputVariable="input"
              name="invService"
              operation="op"
              outputVariable="output"
              partnerLink="serviceName"
              portType="..."/>
      </vxbpel:VPBpelCode>
    </vxbpel:Variant>
    <vxbpel:Variant name="setParams">
      <vxbpel:VPBpelCode>
        <sequence name="setserviceparams">
          <invoke inputVariable="par"
                name="invParams"
                operation="setParams"
                outputVariable="setParamsOut"
                partnerLink="serviceName"
                portType="..."/>
          <invoke inputVariable="input"
                name="invService"
                operation="op"
                outputVariable="output"
                partnerLink="serviceName"
                portType="..."/>
        </sequence>
      <vxbpel:VPBpelCode>
    </vxbpel:Variant>
  </vxbpel:Variants>
</vxbpel:VariationPoint>
```

If one sets parameters on a service by invoking a different operation altogether or altering the type of message sent, it can be modeled in the same way as in the previous section.

3.2.3. System composition

System composition, or more in general, service fragments, can be modeled with VxBPEL as well. Suppose we have the following composition or fragment, which uses the parallel execution container `flow`:

```
<flow name="flow1">
  <links>
    <link name="A"/>
    <link name="B"/>
  </links>
  <invoke ...>
    <sources>
      <source linkName="A">
        <transitionCondition>
          ...
        </transitionCondition>
      </source>
      <source linkName="B">
        <transitionCondition>
          ...
        </transitionCondition>
      </source>
    </sources>
  </invoke>
  <invoke ...>
    <targets>
      <target linkName="A"/>
    </targets>
  </invoke>
  <invoke ...>
    <targets>
      <target linkName="B"/>
    </targets>
  </invoke>
</flow>
```

and we want to define a variant with a sequential container, `sequence`:

```
<sequence name="sequence1">
  <invoke .../>
  <if name="...">
    <condition>...</condition>
    <then>
      <invoke.../>
    <elseif>
      <condition>...</condition>
      <invoke .../>
    </elseif>
  </if>
</sequence>
```

one can then define a variation point with these fragments as variants:

```
<vxbpel:VariationPoint
        name="fragmentExample">
  <vxbpel:Variants>
    <vxbpel:Variant name="flow">
      <vxbpel:VPBpelCode>
        <flow name="flow1">
          <!-- here goes the rest of the
                flow code... -->
        </flow>
      </vxbpel:VPBpelCode>
    </vxbpel:Variant>
    <vxbpel:Variant name="sequence">
      <vxbpel:VPBpelCode>
        <sequence name="sequence1">
          <!-- here goes the rest of the
               sequence code... -->
        </sequence>
      <vxbpel:VPBpelCode>
    </vxbpel:Variant>
  </vxbpel:Variants>
</vxbpel:VariationPoint>
```

In this way, it allows service fragments or service composition changes.

Though some variation points in services can also be enabled through the use of BPEL native constructs such as flows with a transition condition based on a configuration parameter, which may actually be preferred in some cases, the advantage of VxBPEL is that the choices (variants) for the variation point are dynamic. The VxBPEL extension was designed so new variants can be introduced at run-time. This allows for dynamic changes to existing variation points in a VxBPEL process, which is a major advantage over static configuration-based switch-like behaviour, having only a fixed set of choices.

3.3. Prototype

In this section, we discuss how to extend an existing engine to support VxBPEL. It serves two purposes. One is to test the feasibility of VxBPEL. The other is to find a way to allow the management of the variability offered by VxBPEL to be reconfigured externally. In this study,

we have selected ActiveBPEL [1] as the engine, because of its wide acceptance in the Web services community. ActiveBPEL is a freely available, open source, commercial-grade BPEL engine written in Java. It can be run in any Java servlet container, such as Apache Tomcat (the application used for the implementation).

There are two ways of incorporating variability into BPEL processes. The first one is to separate variability definition from the process definition. This can be done either by adding it as a separate part in the process definition file, or by putting it in a separate file. We can add variability inline without affecting the original BPEL file by using a different namespace (http://vxbpel.rug.org, prefix `vxbpel` as illustrated in Fig. 4). When we wish to add variability constructs discussed in the previous sections as a separate part of the file, it will be necessary to indicate which points in the XML file are considered variation points. XPath provides the functionality needed to make this possible. This ensures that standard interpreters of the BPEL file will ignore the elements added as well as everything contained in these elements.

The second one is to define variability inline in the process definition. BPEL elements are enclosed by elements from a different namespace (the `vxbpel` namespace in our case). In this way, it is possible to see which parts of the process are variable by looking at their definitions. Unfortunately, this means the BPEL file can no longer be interpreted by a BPEL engine "as is", as the elements from another namespace, including their children, are ignored by an interpreter. However, a simple transformation using, for example, XSLT would solve this. To indicate that a part of a BPEL process is a variation point, it is enclosed by a ⟨`vxbpel:VariationPoint`⟩ element. Variants defined for this variation point are listed within such an element by a ⟨`vxbpel:Variant`⟩ element. The ⟨`vxbpel:VPBpelCode`⟩ element contains the BPEL elements associated with the enclosing variant.

The advantages of the first way include the following. The original process definition is not altered in any way. It can be executed by any BPEL engine that ignores tags from a different namespace, as defined in the BPEL4WS 1.1 [4] and WS-BPEL 2.0 [21] specification. All the information related to variability is together in one place, presenting a good overview in theory. However, one can imagine that when a process has upwards of 20 variation points, these being defined in XML will not make it easy to read. The disadvantages include that indicating a node by XPath can be error-prone, it is difficult for looking at the complex process definition to determine which parts are currently considered variation points, and the need to duplicate code in case the process definition itself should be executable by any BPEL engine – which also makes maintaining these processes more difficult than necessary.

A big advantage of the second way is that the variability information is located inside the process definition, which makes defining a process as well as implementing a parser or reader for this variability information easier. Also, by looking at the process definition, it is significantly easier for one to see the variability in the process. A disadvantage is that extending BPEL like this makes new process definitions incompatible with the BPEL format and it will no longer be possible for standard engines to read the definition. However, this is also an advantage – if variability is explicitly modeled, it might not desirable at all to be able to execute it regardless. Also, it is possible to transform the process using an XSLT to conform to the standard BPEL format once more, should one really need to execute the process on a standard BPEL engine.

Based on the above comparison, we decided to use the second way (namely the inline approach) to represent variability into BPEL processes, because

- It does not require code duplication, which is error-prone.
- It is easier to implement and to define processes manually, as there are no tools for defining processes with variability.
- Most of all, it is less complicated when parsing, so less time-consuming.

In order to allow ActiveBPEL to execute VxBPEL, two things need to be done. Firstly, the engine must be adapted to recognize and store the new elements introduced when reading in a process definition. Secondly, a definition of behaviour during execution needs to be defined for these elements.

The ActiveBPEL engine, when reading in a process definition, creates a data structure in memory which is similar to a parse tree. This data structure is actually not much more than a blueprint. When a process is invoked, this data structure is consulted to create a new executable data structure based on this blueprint, which is then executed. A parallel invocation of the process will result in multiple of these executable data structures to exist. Fig. 5 depicts this graphically.

The in-process (lower-level) variation points are stored as new elements inside the process' data structure. They are treated in the same way as any standard BPEL activity, up to the point of execution. At this point, a choice needs to be made of which variant's code is to be executed. Fig. 6 shows a graphic representation of the variation point/variant structure.

The choice between variants is determined by the current configuration of the process. This configuration is determined by the state of the configurable (high-level) variation points. These configurable variation points are stored in a new data structure, along with the current configuration. This configuration is accessible for every lower-level variation point in order to determine which of its variants is currently selected. This is graphically depicted in Fig. 7.

In order for the implementation to be meaningful, it is necessary to allow changes to be made to this configuration and thus reconfigure the process itself. In order to allow

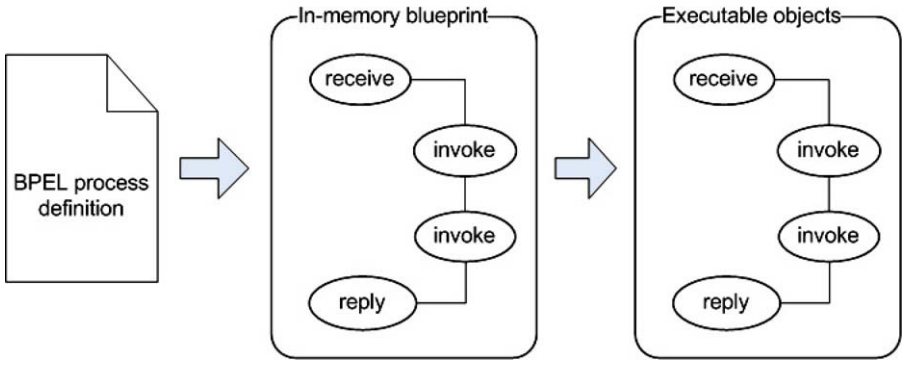

Fig. 5. Graphic representation of process execution in ActiveBPEL.

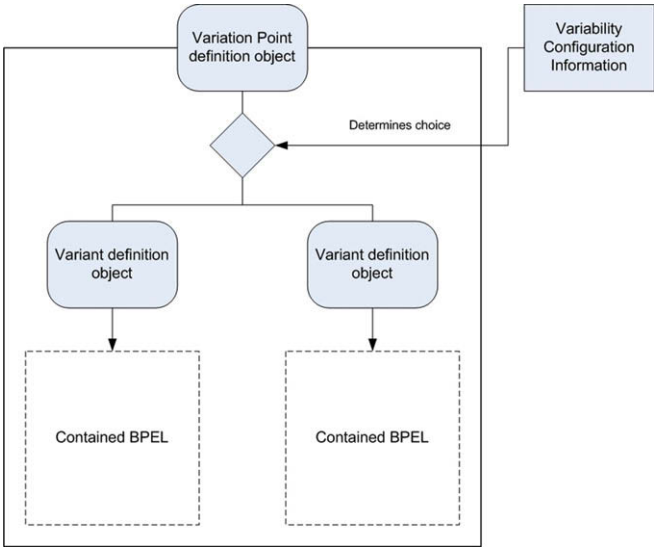

Fig. 6. The variation point and variant structure.

this at run-time, the data structure associated with configurable variation points and the process' configuration exposes certain functionality through JMX [25]. JMX is an extension to Java, allowing Java objects to expose certain functionality (possibly to external tools) allowing management of these objects. By using this exposed functionality to change the state of the configuration-related objects, it is possible to reconfigure the process, even from an external tool.

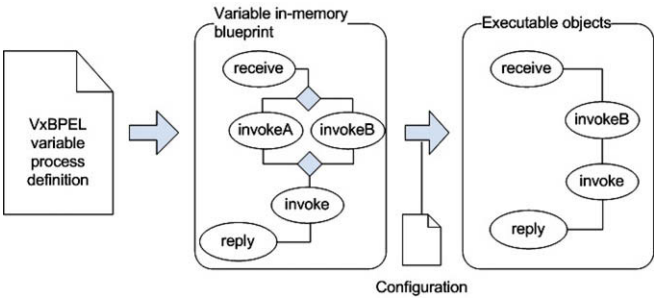

Fig. 7. Graphic representation of variable process execution in the prototype.

It should be noted that the prototype as implemented had several limitations. For example, applying changes to an executing process is not possible. Each instantiation of a process can be different, but once an instance is created, it will use the process definition that was current at instantiation time. This choice was made to keep the complexity of the prototype to a minimum, as it was originally started as a feasibility study. This also allowed us to keep data dependencies and instance migration between process configurations out of scope.

Performance overhead for this extension should be minimal, as only two things have changed. One is the way definitions are read in. This is done only once per deployed process and the amount of time needed to parse a business process definition has only increased because of the increase in the number of activities to be read in per definition. The amount of computation time needed for a variable business process with n activities should therefore not deviate significantly from a non-variable business process with n activities. Variable business process definitions will always contain the configuration information and this increases the parsing time by an amount proportional to the amount of configuration information contained by the process.

Since our extensions are seamlessly integrated into the BPEL engine and the interpretation of variability constructs is analogous to standard constructs in the BPEL specification, the overhead is negligible.

The other change is the switch-like behaviour which is now added to the invocation logic. This is determined by a series of get-operations on Java Hashmap objects once per invocation. Its impact on run-time performance is linearly proportional to the amount of variation points present in the variable business process. This is validated by the case study reported later in the paper from which we did not observe a significant difference in performance.

4. Case study

In this section, we will use a loan approval system to examine VxBPEL and its corresponding supporting platform. The process is taken directly from the WS-BPEL

2.0 specification [21]. It is a simple loan approval Web Service where customers can send their requests for loans. Customers of the service send their loan requests, including personal information and the amount being requested. Using this information, the loan service runs a simple process that results in either a "loan approved" message or a "loan rejected" message.

The approval decision can be reached in two different ways, depending on the amount requested and the risk associated with the requester. For low amounts (less than $10,000) and low-risk individuals, approval is automatic. For high amounts or medium and high-risk individuals, each credit request needs to be studied in greater detail.

To process each request, the loan service uses the functionality provided by two other services. In the streamlined processing available for low amount loans, a "risk assessment" service is used to obtain a quick evaluation of the risk associated with the requesting individual. A full-fledged "loan approval" service (possibly requiring direct involvement of a loan expert) is used to obtain in-depth assessments of requests when the streamlined approval process does not apply.

Testing was done by a small Java application which simulates a client invoking the VxBPEL process that is usually defined by the BPEL process developer. Results of deployment, execution and reconfiguration were verified through a webtool called ActiveBPEL Administration which is bundled with ActiveBPEL. This webtool allows users to see the structure of deployed and executed processes from a browser application. Also, the JMX functionality as exposed by the adaptation of the ActiveBPEL engine could be accessed through a small browser tool that is bundled with the JMX implementation used for testing. This implementation of JMX is called MX4J [20], and the tool used, MX4J/Http Adaptor, exposes the JMX functionality through a relatively simple browser interface. Fig. 8 shows functionality exposed by a configurable variation point. Figs. 9 and 10 show the results of executing the default configuration and the alternative configuration in the ActiveBPEL Administration tool. In the graphic representation of an executed process, activities are shown together with a progress indicator. These indicators are checkmarks (successfully executed activities), crosses (failed or faulted activities), diagonally striked-through circles (non-executed activities, the path through the process did not have to execute this activity) and triangles pointing to the right (currently executing activities).

The implementation was tested with several different cases, each testing a different part of required behaviour. The test cases were made to show the following.

- A deployed VxBPEL process was invoked.
- The configuration of a deployed VxBPEL process was changed using the JMX tool.
- A deployed process' configuration was changed and the process was then invoked (it means that different variants for a variation point were invoked).
- The process configuration remains consistent during execution, even when changes are made to the configuration. To this end, firstly a process with an infinite loop in one variant and a statement to break the loop in another variant was deployed. While the process is in the infinite loop, the configuration was changed so the loop was replaced with the loop-breaking statement. Secondly, a process with a statement executing for a very long time is deployed, with differences in the statements following the long-running statement. Configuration is changed during the long-running statement. This is an important test, as changes during

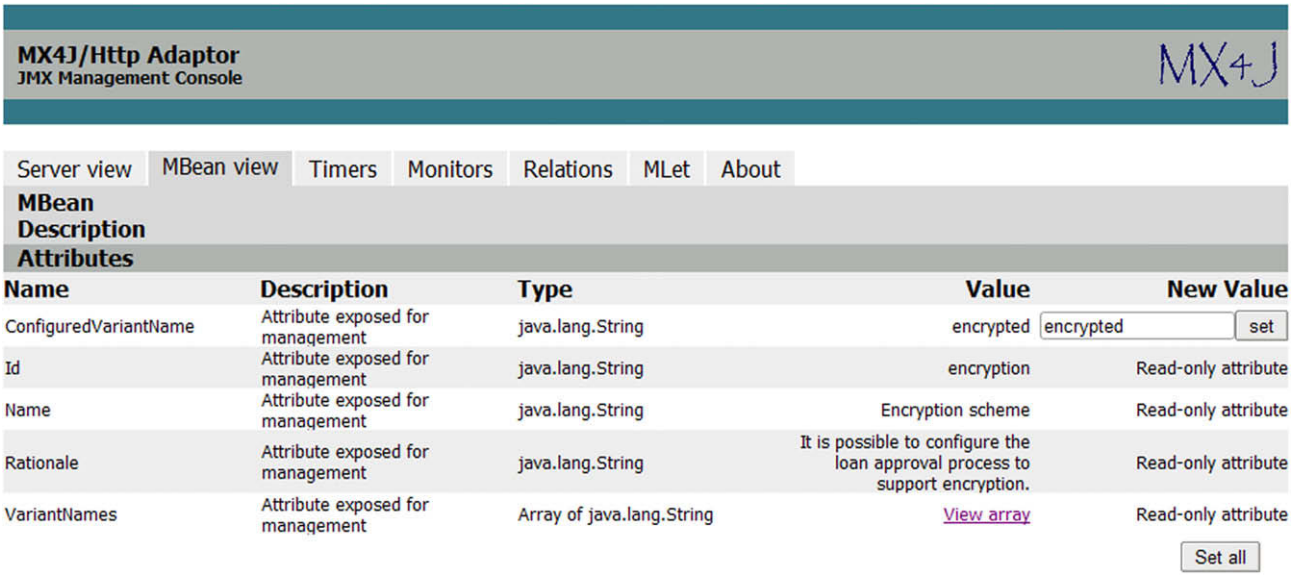

Fig. 8. The JMX HTTP Adaptor webpage where the configuration can be altered.

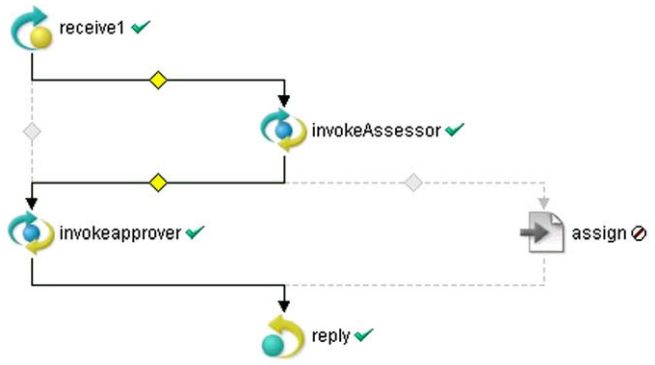

Fig. 9. The standard configuration after execution.

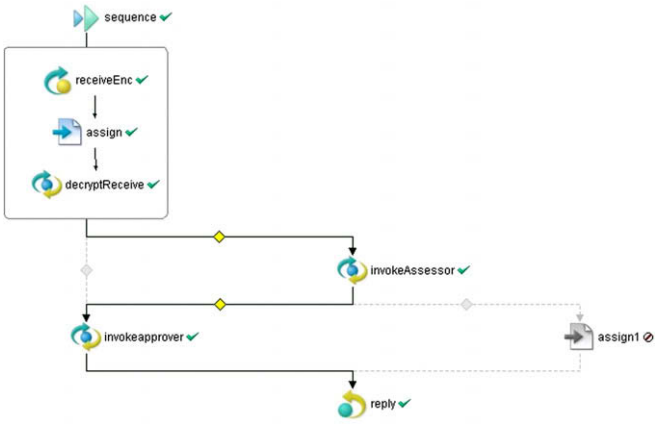

Fig. 10. The alternative configuration after execution.

process execution were declared out of scope (see Section 3.3) and thus changes should not be propagated.

The first four cases were tested using a single process definition. Both Figs. 9 and 10 show a path through the process where the process was successfully executed from receive to reply, showing that this process was correctly and fully executed, the former depicting the process as described and the latter depicting a variation which has a decryption phase as a first step (to handle an encrypted message).

Testing the above cases resulted in the following conclusions:

- It is possible to deploy a VxBPEL process to the implementation.
- It is possible to successfully run the default configuration of such a process.
- It is possible to reconfigure the process to allow future invocations to use the new configuration.
- It is possible to successfully run any reconfiguration of the process.
- Reconfiguring the process definition while an instance of the process is being executed does not affect the running instance, only new instances.

5. Related work

COVAMOF [24] (ConIPF Variability Modeling Framework) is a variability modeling approach that is able to model variability on several levels of abstraction and has explicit modeling for several types of variation. It models variability generically, allows to manage complexity of variability and enables automation in the variability process. The focus of COVAMOF is on variability management in software product lines and as such it does not address modelling variability in the context of Web services directly. The differences between SC systems and component-based systems (such as software product lines) make it difficult or impossible to use COVAMOF (and other approaches) directly for SC systems. Although some work [26] exists that addresses modelling variability in Web services, our approach is the first that is complete enough to enable variability modelling in compositions such as SC systems together with a working prototype.

Several attempts have been reported on extensions to BPEL [7–9,12–16]. We introduce below those addressing the adaptation of BPEL processes, which are closely related to our work.

TRAP/BPEL [16] is a framework that adds autonomic behavior into existing BPEL processes. It aims to make an aggregate web service continue its function even after one or more of its constituent Web services have failed. It assumes that BPEL is used to compose the aggregate web services from the single web services. The framework is developed to monitor the invocation of their partner Web services at runtime. In detail, the framework monitor events such as faults and timeouts from within the adapted process which is augmented with a generic proxy that replaces failed services with predefined or newly discovered alternatives. RobustBPEL-1 [14] and RobustBPEL-2 [15] use static and dynamic proxies, respectively. They are specific, which indicates that a proxy has to be generated for every process, while TRAP/BPEL develops a generic proxy to improve the performance of the previous versions. TRAP/BPEL and its previous versions are a family of extensions to BPEL for enhancing the robust Web services compositions described by BPEL processes. These methods treat the adaptation of Web services compositions implicitly and achieve it only in the level of implements at runtime. Their methods extend neither the BPEL language nor its engine, while they do need the realization of proxy, and cause extra versions of BPEL processes. Our method extends the BPEL language itself and addresses the adaptation both at design-time and at runtime.

wsBus [13] is a Web services message bus middleware which is developed to address QoS concern of Web service compositions. The wsBus introduces the concept of a virtual endpoint where a policy may be attached. Handler bound to the virtual endpoint intercepts request and response messages during the process enactment. All request messages are sent to the virtual point and wsBus

redirects messages to real services. Selection of services is based on monitoring data or QoS metrics. In this way, the approach separates functional requirements (business logic) and nonfunctional requirements (such as QoS). wsBus can provide the optimized QoS during Web services compositions. However, the wsBus may become a bottleneck since a large number of messages are routed through it. Similar to RobustBPEL or TRAP/BPEL, wsBus is a kind of broker which improves QoS by selecting appropriate services for execution at runtime. wsBus focuses on runtime adaptation in terms of Web service composition instances, and adaptation is achieved at a much lower messaging layer at runtime, while our work treats and addresses adaptation at the process specification layer and provides the generic constructs for addressing and specifying adaptation.

AdaptiveBPEL [12] is a framework which aims to support the development of differentiated and adaptive Web services compositions. The concept of aspects originally from Aspect Oriented Software Development is introduced to specify and implement non-functional concerns, such as QoS. The adaptation process is driven by policy, and a policy mediator is used to negotiate a composite policy and oversee the aspects weaving to enforce the negotiated policy. To achieve adaptation of enactment processes, a runtime aspects weaving middleware is integrated on top of a BPEL engine. It is not clearly discussed how the middleware and the BPEL engine interact. The approach addresses the adaptation from the perspective of middleware. It leverages aspect oriented programming techniques to combine concerns which are separately specified in BPEL processes and aspects. It is based on specific BPEL process instances and implements adaptive web service compositions in the implementation level at runtime. The approach also needs extensions to the existing Web service composition platforms, such as ActiveBPEL.

AO4BPEL [8,9] is an aspect-oriented extension to BPEL which addresses the limitation of modularity in current BPEL versions. In AO4BPEL, the business logic is treated as the main concern in workflows, while crosscutting concerns, such as data validation and security, are specified using workflow aspects in a modular way. A prototype implementation of AO4BPEL is presented as a proof-of-concept for aspect-oriented workflow languages which provide concepts of crosscutting modularity such as aspects, join points, pointcuts and advice [7]. The approach addresses the adaptation from the perspective of adaptive workflows. Similar to AdaptiveBPEL [12], the approach proposes to solve the modularity problems with the BPEL using the aspect-oriented concepts in the context of workflow languages. The specifications for business logic and crosscutting concerns are separately specified in different files, which provide better modularity and dynamics. However, in order to support the execution of separate process specifications, they need to be weaved at compile time or at runtime. There are two ways for this task [7]: One is process transformation, which will cause two versions of the workflow processes (one before and one after weaving) have to be maintained; the other is aspect-aware BPEL engine, which needs to modify BPEL engine to support for aspects before and after executing each activity. Since aspect definitions split up the process logic over many different files, this could make debugging a faulty process a difficult task.

There have been several approaches in service-related research to allow for reconfiguration of running processes. Some of these approaches [27] address automatic service substitution in case of failures. They do not allow for a composition's behaviour to be changed apart from a substitution of a single service (and possibly a limited amount of rebinding and replanning needed to support this substitution). Our approach has no support for automatic substitution and does not allow dynamic discovery of alternatives. However, given the flexibility with which variation points can be modelled in our approach, an extended implementation would be able to support both these features as well, as well as offer such automation with regards to compositions instead of merely services.

Other approaches [5,10,23,28] seek to permit automatic composition or automatic reconfiguration of composition of services. This is similar to our approach in that reconfiguration of compositions is central to our approach. As mentioned, automated reconfiguration is not yet possible, but the way our approach models variability does not prohibit such a feature. However, these approaches generally address the problem of a particular part of the system misbehaving and being reconfigured, instead of dynamically changing features supported by the system as VxBPEL allows.

Our approach is unique in that it not only addresses the possibility of single services being replaced, but also allows processes to be reconfigured in significant ways, such as switching between configurations with and without encryption within one single process, with the ability to define many more subtly or significantly different configurations as well. Basically, this means one could capture a family of related processes in one process definition, while keeping the ability to reconfigure this process to any of the family's processes for possibly each request.

6. Conclusion and future work

We have presented VxBPEL, an extension to the BPEL language allowing variability of a service-based system to be modelled. We have developed a prototype to support deployment, execution and reconfiguration of variable processes. Our experiments have validated that it supports changes between each invocation of deployed processes by means of a manual change in the processes' configuration via the variability management interface the implementation exposes.

VxBPEL allows one to capture variation points, variants and realization relations between these variation points. Defining this variability information gives a process interesting capabilities, such as being able to arbitrarily

switch between different levels of QoS by defining variants for each level of QoS. Also, modelling variability concepts as generically as presented means it is possible to capture a family of processes within one process definition, and due to the flexibility of service-based systems, it is possible to switch between these family members at run-time.

Future work includes support for more types of variation points (optional and open) and allowing running processes' configurations to change, which would require more extensions to BPEL and definition of rules for behaviour of a process when variation points are reconfigured in a running process. This would also require addressing the issues of data dependencies and instance migration between process configurations.

Acknowledgements

This work is partially supported by the EU-funded project SeCSE (IST Contract No. 511680) and the Science and Technology Foundation of Beijing Jiaotong University (Grant No. 2007RC099). The authors appreciate the anonymous reviewers for their invaluable comments which helped to greatly improve the presentation of the paper.

References

[1] Active Endpoints, ActiveBPEL engine. Available from: <http://www.activebpel.org>.
[2] F. Bachmann, L. Bass, Managing variability in software architectures, in: Proceedings of the ACM SIGSOFT Symposium on Software Reusability (SSR'01), 2001.
[3] D. Batory, S. O'Malley, The design and implementation of hierarchical software systems with reusable components, ACM Transactions on Software Engineering and Methodology 1 (4) (1992) 355–398.
[4] BEA, IBM, Microsoft, SAP AG, Siebel Systems, Business Process Execution Language for Web Services V1.1 specification (2003). Available from several of the partners' web pages, e.g. <http://www-128.ibm.com/developerworks/library/specification/ws-bpel/>.
[5] B. Benatallah, M. Dumas, Q. Sheng, Facilitating the rapid development and scalable orchestration of composite web services, Distributed and Parallel Databases 17 (1) (2005) 5–37.
[6] J. Bosch, Design & Use of Software Architectures, Addison-Wesley, 2000.
[7] A. Charfi, Aspect-Oriented Workflow Languages: AO4BPEL and Applications, Ph.D. thesis, TU Darmstadt, Fachbereich Informatik (2007). URL <http://elib.tu-darmstadt.de/diss/000852>.
[8] A. Charfi, M. Mezini, Aspect-Oriented Web Service Composition with AO4BPEL, in: ECOWS, vol. 3250 of LNCS, Springer, 2004, pp. 168–182.
[9] A. Charfi, M. Mezini, AO4BPEL: an aspect-oriented extension to BPEL, World Wide Web 10 (3) (2007) 309–344.
[10] M. Colombo, E.D. Nitto, M. Mauri, SCENE: a service composition execution environment supporting dynamic changes disciplined through rules, in: International Conference on Service-Oriented Computing (ICSOC'06), vol. 4292 of Lecture Notes in Computer Science, Chicago, USA, 2006, pp. 191–202.
[11] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, S. Weerawarana, Unraveling the web services web: an introduction to SOAP, WSDL, and UDDI, IEEE Internet Computing 6 (2) (2002) 86–93.
[12] A. Erradi, P. Maheshwari, AdaptiveBPEL: a policy-driven middleware for flexible web services compositions, in: Proceedings of Middleware for Web Services (MWS), 2005.
[13] A. Erradi, P. Maheshwari, wsBus: QoS-aware middleware for reliable web services interaction, in: Proceedings of IEEE International Conference on e-Technology, e-Commerce and e-Service, Hong Kong, China, 2005.
[14] O. Ezenwoye, S. Sadjadi, Enabling robustness in BPEL processes, in: Proceedings of the 8th International Conference on Enterprise Information Systems (ICEIS-06), 2006.
[15] O. Ezenwoye, S. Sadjadi, RobustBPEL-2: Transparent autonomization in aggregate web services using dynamic proxies, Tech. Rep. FIU-SCIS-2006-06-01, School of Computing and Information Sciences, Florida International University, 11200 SW 8th St., Miami, FL 33199, June 2006.
[16] O. Ezenwoye, S. Sadjadi, TRAP/BPEL: A Framework for Dynamic Adaptation of Composite Services, Tech. Rep. FIU-SCIS-2006-06-02, School of Computing and Information Sciences, Florida International University (2006).
[17] Y. Han, A. Sheth, C. Bussler, A taxonomy of adaptive workflow management, in: CSCW-98 Workshop, Towards Adaptive Workflow Systems, 1998.
[18] I. Jacobson, M. Griss, P. Jonsson, Software Reuse. Architecture, Process and Organization for Business Success, Addison-Wesley, 1997.
[19] M. Little, Transactions and web services, Communications of the ACM 46 (10) (2003) 49–54.
[20] MX4J, MX4J homepage. Available from: <http://mx4j.sourceforge.net/>.
[21] OASIS, Web Services Business Process Execution Language Version 2.0 Committee Draft, The latest version of this draft is downloadable through <http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel> (March 2006).
[22] C. Peltz, Web services orchestration and choreography, Computer 36 (10) (2003) 46–52.
[23] J. Siljee, I. Bosloper, J. Nijhuis, D. Hammer, DySOA: making service systems self-adaptive, in: International Conference on Service-Oriented Computing (ICSOC'05), Amsterdam, the Netherlands, 2005, pp. 255–268.
[24] M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch, COVAMOF: a framework for modeling variability in software product families, in: The Third Software Product Line Conference (SPLC 2004), Boston, USA, 2004, pp. 197–213.
[25] Sun Microsystems, Java Management Extensions. Available from: <http://java.sun.com/products/JavaManagement/>.
[26] N. Topaloglu, R. Capilla, Modeling the variability of web services from a pattern point of view, in: Web Services, European Conference, ECOWS 2004, Erfurt, Germany, September 27–30, 2004, Proceedings, vol. 3250 of Lecture Notes in Computer Science, 2004, pp. 128–138.
[27] W. Tsai, W. Song, R. Paul, Z. Cao, H. Huang, Services-oriented dynamic reconfiguration framework for dependable distributed computing, in: 28th Annual International Computer Software and Applications Conference (COMPSAC'04), Hongkong, 2004, pp. 554–559.
[28] K. Verma, K. Gomadam, A. Sheth, J. Miller, Z. Wu, The METEOR-S approach for configuring and executing dynamic web processes, Tech. Rep., LSDIS Lab, University of Georgia, Athens, Georgia, 2005.